Source-Code Instrumentation and Quantification of Events

Robert E. Filman
RIACS
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035 U.S.A.
+1 650–604–1250
rfilman@mail.arc.nasa.gov

Klaus Havelund
Kestrel Technology
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035 U.S.A
+1 650–604–3366
havelund@email.arc.nasa.gov

ABSTRACT
Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions. Varieties of AOP systems are characterized by which quantified assertions they allow, what they permit in the actions of the assertions (including how the actions interact with the base code), and what mechanisms they use to achieve the overall effect. Here, we argue that all quantification is over dynamic events, and describe our preliminary work in developing a system that maps dynamic events to transformations over source code. We discuss possible applications of this system, particularly with respect to debugging concurrent systems.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – aspects. D.3.2 [Programming Languages] Language Classifications – aspect-oriented programming. D.2.3 [Software Engineering] Coding Tools and Techniques. D.2.5 [Testing and Debugging] Debugging aids.

General Terms
Languages.

Keywords
Quantification, events, dynamic events, debugging, program transformation, model checking.

1. INTRODUCTION
Elsewhere, we have argued that the programmatic essence of Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions [10,12]. That is, in an AOP system, one wants to be able to say things of the form, "In this program, when the following happens, execute the following behavior," without having to go around marking the places where the desired behavior is to happen. Varieties of AOP systems are characterized by which quantified assertions they allow, what they permit in the actions of the assertions (including how the actions interact with the base code), and what mechanisms they use to achieve the overall effect. In this paper, we describe our preliminary work in developing a system that takes the notion of AOP as quantification to its logical extreme. Our goal is to develop a system where behavior can be attached to any event during program execution. We describe the planned implementation of this system and discuss possible applications of this technology, particularly with respect to debugging and validating concurrent systems.

2. EVENTS
Quantification implies matching a predicate about a program. Such a predicate must be over some domain. In the quantification/implicit invocation papers, we distinguished between static and dynamic quantification.

Static quantification worked over the structure of the program. That is, with static quantification, one could reference the programming language structures in a system. Examples of such structures include reference to program variables, calls to subprograms, loops, and conditional tests.

Many common AOP implementation techniques can be understood in terms of quantified program manipulation on the static structure of a program. For example, wrapping (e.g., as seen in Composition Filters [1], OIF [11], or AspectJ [19,20]) is effectively embedding particular function bodies in more complex behavior. AspectJ and OIF also provide a call-side wrapping, which can be understood as surrounding the calling site with the additional behavior. An operation such as asserting that class **A**'s use of *x* is the same as class **B**'s use of *y* in Hyper/J [22] can be realized by substituting a reference to a common generated variable for *x* in the text of **A**, and *y* in **B**.

Dynamic quantification, as described in those papers, speaks to matching against events that happen in the course of program execution. An example of dynamic quantification is the jumping-aspect problem [2], where a method behaves differently depending upon whether or not it has been called from within (in the calling-stack sense) a specified routine. Other examples of interesting dynamic events include the stack exceeding a particular size, the fifth unsuccessful call to the login routine with a different password, a change in the number of references to an object, a confluence of variable values (e.g., when $x + y > z$), the blocking of a thread on a synchronization lock, or even a change in the executing thread. The cflow operator in AspectJ is a dynamic quantification predicate.

We are coming to the belief that all events are dynamic. Static quantification should be understood as just the subspecies of events that can be simply inferred, on a one-to-one basis, from the structures of a program. Static quantification is attractive for its straightforward AOP implementation, lower complexity, and independence of programming environment implementation, but unless one starts processing the program comments, there's little

Table 1: Events and event loci

Event	Syntactic locus
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements
Waiting on a lock	Wait and synchronize statements
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could happen anywhere

in the static structure of a program that isn't marked by its dynamic execution.

If the abstract syntax tree is the domain of static quantification, what is the domain of dynamic quantification? Considering the examples in this section, it really has to be events that change the state (both data state and "program counter") of the base language's abstract interpreter. However, defining anything in terms of the abstract interpreter is problematic. First, as was illustrated in Smith's work on 3-Lisp [5], programming languages are not defined in terms of their abstract interpreters. The same language can be implemented with many different interpreters. The set of events generated by one implementation of a language may not correspond to the events generated by another. For example, a run-time environment that manages its own threads is not at all the same as one that relies on the underlying operating system for thread management. Neither is the same as one that takes advantage of the multiple processors of a real multi-processor machine. Second, compilers have traditionally been allowed to optimize—rearrange programs while preserving their input-output semantics. An optimizing compiler may rearrange or elide an "obvious" sequence of expected events. And finally, the data state of the abstract interpreter (including, as it does, all of memory) can be a grand and awkward thing to manipulate.

3. A LANGUAGE OF EVENTS

We view these limitations as bumps in the road, rather than barriers. While we may not be able to capture everything that goes on in a particular interpretive environment, we can get close enough for most practical purposes. The strategy we adopt is to argue that most dynamic events, while not necessarily local to a particular spot in the source code, are nevertheless tied to places in the source code. Table 1 illustrates some primitive events and their associated code loci.

Users are likely to want to express more than just primitive events. The language of events will also want to describe relationships among events, such as that one event occurred before another, that a set of events match some particular predicate, that an event occurred within a particular timeframe, or that no event matching a particular predicate occurred. This suggests that the event language will need (1) abstract temporal relationships, such as "before" and "after," (2) abstract temporal quantifiers, such as "always" and "never", (3) concrete temporal relationships referring to clock time, (4) cardinality relationships on the number times some event has occurred, and (5) aggregation relationships for describing sets of events.

4. SYSTEM ARCHITECTURE

We envision a mechanism where a description of a set of event-action pairs, along with a program, would be presented to a compiler. Each event action pair would include a sentence describing the interesting event in the event language and an action to be executed when that event is realized. Said actions would be programs, and would be parameterized with respect to the elements of the matching events. Examples of such assertions are:

- On every call to method *foo* in a class that implements the interface **B**, replace the second parameter of the call to *foo* with the result of applying method *f* to that parameter.
- Whenever the value of $x+y$ in any object of class **A** ever exceeds *5*, print a message to the log and reset *x* to *0*.
- If a call to method *foo* occurs within (some level down on the stack) method *baz* but without an intervening call to method *mumble*, omit the call to method *gorp* in the body of *foo*.

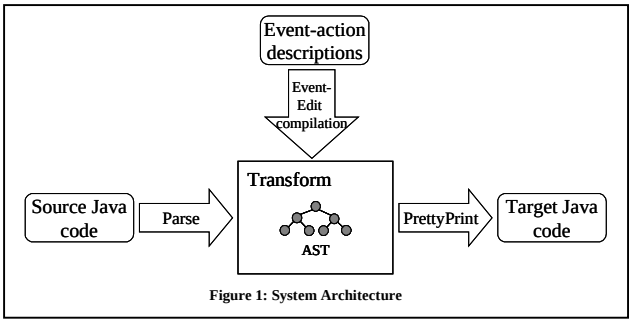

Figure 1: System Architecture

These examples are in natural language. Of course, any actual system will employ something formal.

Clearly, a sufficiently "meta" interpretation mechanism would give us access to many interesting events in the interpreter, enabling a more direct implementation of these ideas. It has often been observed that meta-interpretative and reflective systems can be used to build AOP systems [29]. However, meta-interpreters have traditionally exhibited poor performance. We are looking for implementation strategies where the cost of event recognition is only paid when event recognition is used. This suggests a compiler that would transform programs on the basis of event-action assertions. Such a compiler would work with an extended abstract syntax tree representation of a program. It would map each predicate of the event language into the program locations that could affect the semantics of that event. Such a mapping requires not only abstract syntax tree generation (parsing) and symbol resolution, but also developing primitives with respect to the control and data flow of the program, determining the visibility and lifetimes of symbols, and analyzing the atomicity of actions with respect to multiple threads.

Java compiles into an intermediate form (Java byte codes). In dealing with Java, there is also the choice as to whether to process with respect to the source code or the byte code. Each has its advantages and disadvantages. Byte codes are more real: many of the issues of interest (actual access to variables, even the power consumption of instructions) are revealed precisely at the byte code level. Working with byte codes allows one to modify classes for which one hasn't the source code, including the Java language packages themselves. (JOIE [3] and Jmangler [21] are examples of an AOP systems that perform transformations at the byte code level.) On the other hand, source code is more naturally understandable, allows writing transformations at the human level, and eliminates the need for understanding the JVM and the actions of the compiler. (De Volder's Prolog-based meta-programming system is an example of source-level transformation for AOP [6,7].) We find the complexity arguments appealing. Thus, our implementation plan is to work at the source code level.

5. EXAMPLES

Event quantification is a general framework for supporting aspect oriented programming. It can be used for functionality enhancement, where a program is extended with aspects that add new functionality. For example, a program could be made more reliable by transforming its database update events to also send messages to a backup log. Although functionality extension is a general goal for AOP, we instead discuss some examples within the area of program verification. (In some cases, we expect to be able to extend program behavior for *functionality insurance*: recovering from some classes of program failure.)

In previous work, we studied various program verification techniques for analyzing the correctness of programs. Our work can be classified into two categories: *program monitoring* [17] and *program scheduling* [16,27]. The latter is often called *model checking*.

5.1 Monitoring

Specification-based monitoring consists of monitoring the execution of a program, represented by a sequence of events, by validating the events against a requirements specification. The specification is written in some formal language, typically a temporal logic [24]. For example, a typical requirement is, "Whenever TEMP becomes 100 then within 3 seconds ALARM becomes true." A typical requirement specification has many such assertions. We want to be able to run the program and monitor that specification assertions hold throughout the event trace. The Java PathExplorer system [17] implements this kind of capability. It uses the byte-code engineering tool Jtrek [18] to instrument Java byte code to emit events to an observer, which contains a data structure representing the formulae to be checked. Every event emitted from the running program causes a modification of the data structure. A warning is raised when a specification is violated. We plan to experiment using event quantification at the source code level instead of at the byte code level. The events to be caught are obviously those implicitly referred to in the formula—in the above example, updates to the variables TEMP and ALARM. That is, whenever one of these variables is updated, an event consisting of the variable name, the value, and a timestamp can be emitted to the observer. (The evaluation of the temporal formula can even be performed as part of the quantification action instead of in a separate observer, if real-time performance is not an issue.) Operating on the source code level simplifies creating the instrumentation, as one can work in a high-level language, not byte code. The commercial-available Temporal Rover system performs specification-based monitoring, but does not do automated code instrumentation [8].

Algorithm-based monitoring, like specification-based monitoring, watches the execution of a program emitting events. Rather than matching against user-defined specifications, algorithm-based monitoring uses certain general algorithms for detecting particular kinds of error conditions. Examples are algorithms for detection of deadlock and data race potentials in concurrent programs. These algorithms are interesting since the actual deadlocks or data races do not have to occur in an execution trace in order to be identified as a potential problem. An arbitrary execution trace will normally suffice to identify problems. For example, a cyclic relationship between the locks in a program (thread **T1** takes lock *A* and then *B*, while thread **T2** takes *B* and then *A*) is a potential deadlock. A similar algorithm exists for data races [25]. These algorithms have been implemented in PathExplorer using byte code engineering, and we anticipate trying them out using event quantification.

5.2 Scheduling

Thread scheduling consists of influencing a program's scheduling in order to explore more thread interleavings than would otherwise be achieved with normal testing techniques. As an example, the above mentioned deadlock situation can be explicitly

demonstrated by scheduling the threads such that **T1** takes *A*, and then **T2** immediately takes *B*. Such a schedule might never be seen during normal test of the program. Thread scheduling can be achieved by introducing a centralized scheduler and forcing all threads to communicate with that scheduler when shared data structures (such as locks) are accessed. The scheduler then decides which thread to run, while at the same time keeping track of its scheduling choices. This information can then be used to direct the program to explore new interleavings. We have earlier developed the Java PathFinder system [16, 27] for performing such scheduling analysis using model checking. In order to avoid exploring the reachable subtree below a given program state several times, states are stored in cache, and search is aborted when a state has been visited before. Using quantification, we plan to experiment with state-less model checking [15, 24] where a program's different interleavings are explored, but without storing states. An example of program modification to detect synchronization faults is ConTest [8].

6. RELATED WORK

De Volder and his co-workers [6,7] have argued for doing AOP by program transformation, using a Prolog-based system working on the text of Java programs. We want to extend those ideas to program semantics, combining both the textual locus of dynamic events and transformations requiring complex analysis of the source code.

At the 1998 ECOOP AOP workshop, Fradet and Südholt [13] argued that certain classes of aspects could be expressed as static program transformations. They expanded this argument at the 1999 ECOOP AOP workshop to one of checking for robustness—non-localized, dynamic properties of a system's state [14]. Colcombet and Fradet realized an implementation of these ideas in [4], applying both syntactic and semantic transformations to enforce desired properties on programs. In that system, the user can specify a desired property of a program as a regular expression on syntactically identified points in the program, and the program is transformed into one that raises an exception when the property is violated. Other transformational systems include, *Ku* a notational attempt at formalizing transformation [27], and Schonger et al's proposal to express abstract syntax trees in XML and use XML transformation tools for tree manipulation [26].

Nelson et al. identify three concern-level foundational composition operators: correspondence, behavioral semantics and binding [22]. Correspondence involves identifying names in different entities that are "the same"—for data items, things that should share storage; for functions, functional fragments that need to be assembled into a whole. Behavioral semantics describe how the functional fragments are assembled. Binding is the usual issue of the statics and dynamics of system construction and change. They discuss alternative formal techniques for establishing properties of composed systems within this basis.

Walker and Murphy argue for events as appropriate "join points" for AOP, and that the events exposed by AspectJ are inadequate [32].

7. CONCLUDING REMARKS

In this paper, we've examined the idea of implementing AOP systems as programs transformed by quantified responses to dynamic events. Two comments about the place of such a system in the order of things are worth making:

- We've been talking about implementation environments, not software engineering. An underlying implementation does not imply anything about the "right" organization of "separate concerns" to present to a user. In particular, we have been completely agnostic about the appropriate structure for the actions of action-event pairs. It may be the case that unqualified use of an event language with raw action code snippets is a software engineering wonder, but we doubt it.

- An environment that can map from quantified dynamic events to modified code would be an excellent environment for experimenting with and building systems for AOP. In some sense, these ideas can be viewed as a domain-specific language for developing aspect-oriented languages.

8. ACKNOWLEDGMENTS

Our thanks to Tarang Patel and Tom Pressburger for their comments on the draft of this paper.

9. REFERENCES

[1] Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 51–57.

[2] Brichau, J., De Meuter, W., and De Volder, K. Jumping aspects. Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, Jun. 2000. http://trese.cs.utwente.nl/Workshops/adc2000/papers/Brichau.pdf

[3] Cohen, G. Recombining concerns: Experience with transformation. First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (OOPSLA '99), Oct. 1999, www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws23-cohen.pdf

[4] Colcombet, T. and Fradet, P. Enforcing trace properties by program transformation. *Proc. 27th ACM Symp. Principles of Programming Languages*, Boston, Jan. 2000, pp. 54–66.

[5] des Rivieres, J. and Smith, B. C. The implementation of procedurally reflective languages. *Conf. Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984, pp. 331–347.

[6] De Volder, K., Brichau, J., Mens, K., and D'Hondt, T. Logic meta-programming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf

[7] De Volder, K., and D'Hondt, T. Aspect-oriented logic meta programming. *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616*, Springer-Verlag, 1999, pp. 250–272.

[8] Drusinsky, D. The Temporal Rover and the ATG Rover. *SPIN Model Checking and Software Verification, LNCS* 1885, K. Havelund, J. Penix, and W. Visser (Eds.) Springer, 2000, pp. 323–330.

[9] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S. Multithreaded Java program test generation. *IBM Systems Journal, Vol. 41*, No. 1, 2002, pp. 111–125.

[10] Filman, R.E. What is aspect-oriented programming, revisited. Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming, Budapest, Jun. 2001. http://trese.cs.utwente.nl/Workshops/ecoop01asoc/papers/Filman.pdf

[11] Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting ilities by controlling communications. *Comm. ACM, Vol. 45,* No. 1, Jan. 2002, pp. 116–122.

[12] Filman, R. E. and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Oct. 2000. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf

[13] Fradet, P. and Südholt, M. Towards a generic framework for aspect-oriented programming, Third AOP Workshop, *ECOOP'98 Workshop Reader, LNCS, 1543*, pp. 394–397, Jul. 1998. trese.cs.utwente.nl/aop-ecoop98/papers/ Fradet.pdf

[14] Fradet, P and Südholt, M. An aspect language for robust programming. Int. Workshop on Aspect-Oriented Programming, ECOOP, Jun. 1999. http://trese.cs.utwente.nl/aop-ecoop99/papers/fradet.pdf

[15] Godefroid P. Model checking for programming languages using VeriSoft. *Proc. of 24th ACM Symp. on Principles of Programming Languages*, Paris, Jan. 1997, pp. 174–186.

[16] Havelund K. and Pressburger T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, Vol. 2*, No. 4, Apr. 2000, pp. 366–381.

[17] Havelund K. and Rosu, G. Monitoring Java programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01), Electronic Notes in Theoretical Computer Science, Vol. 55*, No. 2, Elsevier Science, Paris, Jul. 2001.

[18] Jtrek. Compaq. http://www.compaq.com/java/download/jtrek

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ, *Proceedings ECOOP 2001*, J. L. Knudsen (Ed.) Berlin: Springer-Verlag LNCS 2072, pp. 327–353.

[20] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting started with AspectJ. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 59–65.

[21] Kniesel, G., Costanza, P., and Austermann, M. JMangler–A Framework for Load-Time Transformation of Java Class Files. Proc. First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001*)*, Florence, Nov. 2001, http://www.informatik.uni-bonn.de/~costanza/SCAM_jmangler.pdf

[22] Nelson, T., Cowan, D. and Alencar, P. Supporting formal verification of crosscutting concerns. *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, Reflection 2001*, A. Yonezawa and S. Matsuoka (Eds.) Sep. 2001, Kyoto, Berlin: Springer-Verlag, LNCS 2192, pp. 153-169.

[23] Ossher, H. and Tarr, P. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 43–50.

[24] Pnueli A. The temporal logic of programs. *Proc. 18th IEEE Symp. Foundations of Computer Science*, 1977, pp. 46–57.

[25] Savage S., Burrows M., Nelson G., Sobalvarro P., and Anderson T. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems, Vol. 15*, No. 4, Nov. 1997.

[26] Schonger, S., Pulvermueller, E., and Sarstedt, S. Aspect oriented programming and component weaving: using XML representations of abstract syntax trees. Workshop Aspektorientierte Softwareentwicklung, Institut für Informatik III, Universität Bonn, Feb. 2002, i44w3.info.uni-karlsruhe.de/~pulvermu/workshops/aosd2002/submissions/schonger.pdf.

[27] Skipper, M. A Model of composition oriented programming, Proc. Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Int'l Conf/ on Software Engineering, Limerick, Ireland, June 2000, www.research.ibm.com/hyperspace/workshops/icse2000/Papers/skipper.pdf

[28] Stoller S. D. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, in press.

[29] Sullivan, G. T. Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 95–97.

[30] Teitelman, W. and Masinter, L. The Interlisp programming environment. *Computer Vol. 14*, No. 4, 1981, pp. 25–34.

[31] Visser W., Havelund K., Brat G., and Park S. Model checking programs. *Proc. ASE'2000: The 15th IEEE Intl. Conf. Automated Software Engineering*, Sep. 2000, pp. 3–12.

[32] Walker, R. J. and Murphy, G. C. Joinpoints as ordered events: towards applying implicit context to aspect-orientation. Workshop on Advanced Separation of Concerns in Software Engineering at ICSE, Toronto, May, 2001, www.research.ibm.com/hyperspace/workshops/icse2001/Papers/walker.pdf.